Lecture 20
Principles of Functional Programming
Summer 2020

Sequences and Parallelism II

Section 1

The Sequence Library

The List-Tree Duality

- Lists are convenient because their elements are in linear order, and can be easily indexed
- Trees are nice because they more easily support parallelism

Can we have a data structure which combines the better features of both?

seq

- We've defined a signature `SEQUENCE`, containing an abstract type `'a seq` and a variety of operations on `seq`s.
- We've implemented `Seq :> SEQUENCE` such that the functions meets the bounds specified in the documentation
- How's it implemented? Who cares?
- By analogy to lists, we'll write sequences as

$$\langle 1,\ 3,\ {\sim}7,\ 2,\ 6,\ 4\rangle : \texttt{int Seq.seq}$$

This is a mathematical notation, *not* SML syntax.

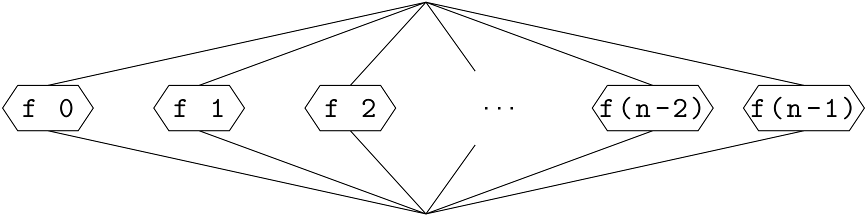
20.1

```
val empty : unit -> 'a seq
val singleton : 'a -> 'a seq
val fromList : 'a list -> 'a seq
val tabulate : (int -> 'a) -> int -> 'a seq
```

20.2

```
val nth : 'a seq -> int -> 'a
val null : 'a seq -> bool
val length : 'a seq -> int
val toList : 'a seq -> 'a list
val toString : ('a -> string) -> 'a seq ->
  string
val equal : ('a * 'a -> bool) -> 'a seq * 'a
    seq -> bool
```

tabulate cost graph



$$W_{\texttt{tabulate}}(n) = \sum_{\texttt{i=0}}^{\texttt{n-1}} W_{\texttt{f(i)}}$$

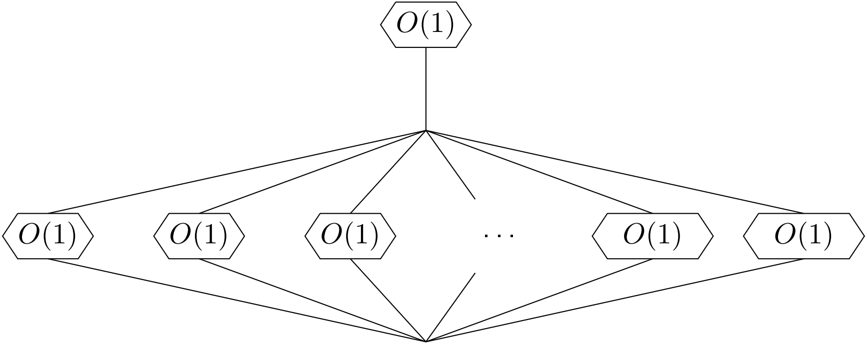
$$S_{\texttt{tabulate}}(n) = \max_{\texttt{i=0}}^{\texttt{n-1}} S_{\texttt{f(i)}}$$

`rev : 'a Seq.seq -> 'a Seq.seq`
REQUIRES: true
ENSURES: `rev S` evaluates to a sequence `S'` containing the exact same elements as `S`, in reverse order

20.5

```
fun rev S =
  let
    val n = Seq.length S  (* O(1) *)
  in
    (* O(n) work, O(1) span *)
    Seq.tabulate (fn i => Seq.nth S (n-i-1)) n
  end
```

Analysis



$$W_{\texttt{rev}}(n) = O(n) \qquad S_{\texttt{rev}}(n) = O(1)$$

Using tabulate

```
append : 'a Seq.seq * 'a Seq.seq -> 'a Seq.
seq
```
REQUIRES: true

ENSURES: `append(S1,S2)` evaluates to a sequence `S'` containing the elements of `S1`, followed by the elements of `S2`

Using tabulate

20.6

```
fun append(S1,S2) =
  let
    (* O(1) *)
    val m = Seq.length S1
    val n = Seq.length S2

    (* O(1) *)
    fun f i =
      case i<m of
        true => Seq.nth S1 i
      | false => Seq.nth S2 (i-m)
  in
    (* O(n+m) work, O(1) span *)
    Seq.tabulate f (m+n)
  end
```

```
val filter
  : ('a -> bool) -> 'a seq -> 'a seq
val map
  : ('a -> 'b) -> 'a seq -> 'b seq
val reduce
  : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
val reduce1
  : ('a * 'a -> 'a) -> 'a seq -> 'a
val mapreduce
  : ('a -> 'b) -> 'b -> ('b * 'b -> 'b)
    -> 'a seq -> 'b
```

Seq.filter p ⟨x0, x1, x2, ..., x(n-1)⟩

$$W = \sum_{\texttt{i=0}}^{\texttt{n-1}} W_{\texttt{p(xi)}} + k\log(\texttt{n})$$

$$S = \max_{\texttt{i=0}}^{\texttt{n-1}} S_{\texttt{p(xi)}} + k\log(\texttt{n})$$

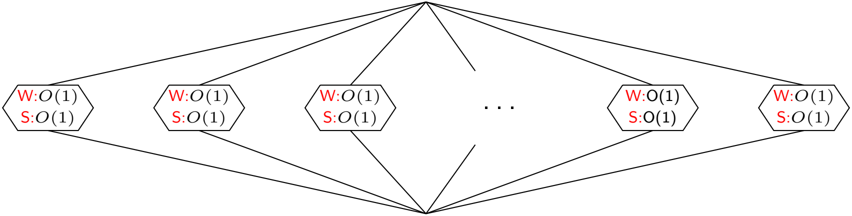
```
val filter
  : ('a -> bool) -> 'a seq -> 'a seq
val map
  : ('a -> 'b) -> 'a seq -> 'b seq
val reduce
  : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
val reduce1
  : ('a * 'a -> 'a) -> 'a seq -> 'a
val mapreduce
  : ('a -> 'b) -> 'b -> ('b * 'b -> 'b)
    -> 'a seq -> 'b
```

Seq.map f $\langle$x0, x1, x2, ..., x(n-1)$\rangle$

$$W = \sum_{\texttt{i=0}}^{\texttt{n-1}} W_{\texttt{f(xi)}}$$

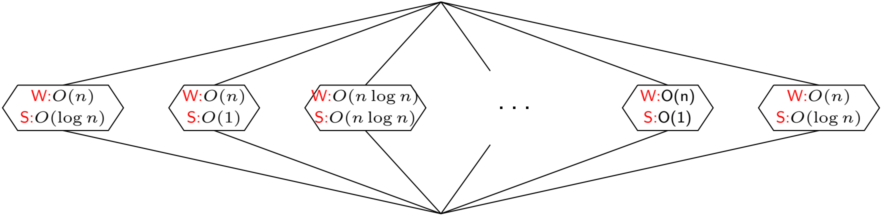
$$S = \max_{\texttt{i=0}}^{\texttt{n-1}} S_{\texttt{f(xi)}}$$

Mapping with a constant-time function



$$W = O(n) \qquad S = O(1)$$

Mapping with a non-constant-time function



$$W = O(n^2) \qquad S = O(n \log n)$$

20.3

```
val filter
  : ('a -> bool) -> 'a seq -> 'a seq
val map
  : ('a -> 'b) -> 'a seq -> 'b seq
val reduce
  : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
val reduce1
  : ('a * 'a -> 'a) -> 'a seq -> 'a
val mapreduce
  : ('a -> 'b) -> 'b -> ('b * 'b -> 'b)
    -> 'a seq -> 'b
```

reduce

```
reduce : ('a * 'a -> 'a) -> 'a list -> 'a Seq
.seq -> 'a
```

REQUIRES: g is total and associative: for all a, b, c,

$$g(g(a,b),c) \cong g(a,g(b,c))$$

ENSURES:

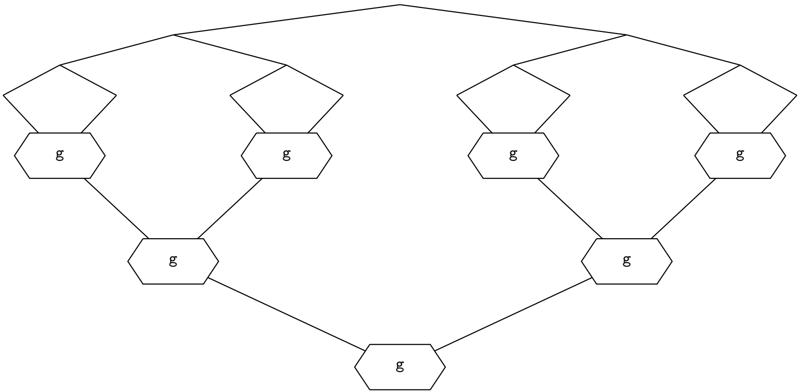
$$\texttt{Seq.reduce g z S} \cong \texttt{foldr g z (Seq.toList S)}$$

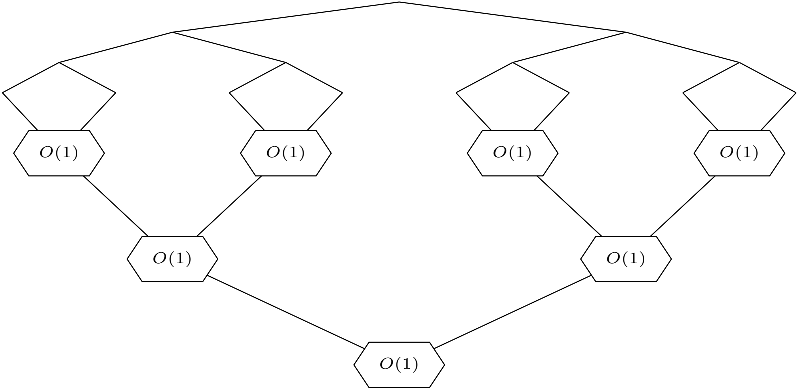
reduce g z ⟨x1,x2,x3,x4,x5,x6,x7⟩

≅ g(x1,g(x2,g(x3,g(x4,g(x5,g(x6,g(x7,z)))))))

≅ g(g(g(x1,x2), g(x3,x4)), g(g(x5,x6),g(x7,z)))

reduce cost graph



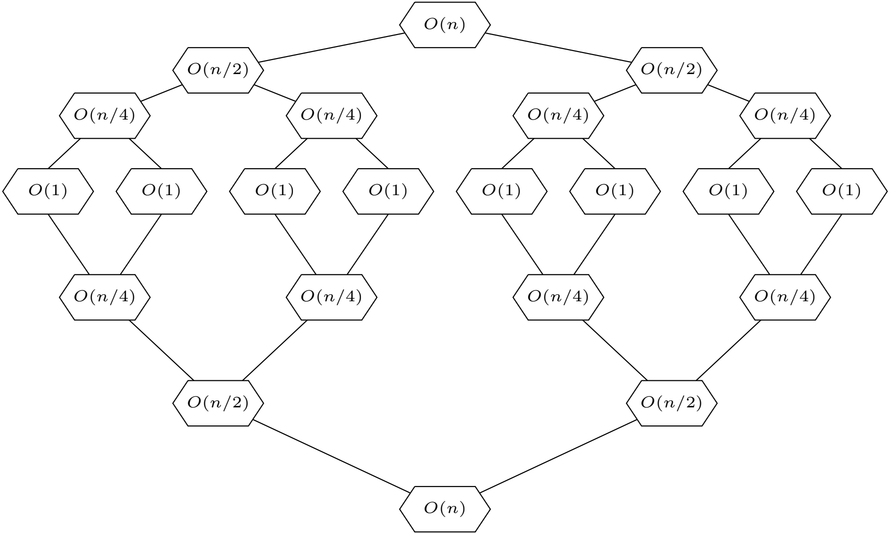


$$W = O(n) \qquad S = O(\log n)$$

20.7

```
val sum = Seq.reduce op+ 0
```

List `msort` work/span (assuming `cmp` $O(1)$)



Sorting using reduce

20.4

```
val sort
  : ('a * 'a -> order) -> 'a seq -> 'a seq
val merge
  : ('a * 'a -> order) -> 'a seq * 'a seq ->
  'a seq
```

Assuming `cmp` is $O(1)$, then `merge cmp (S1,S2)` has runtime

$$W(m,n) = O(m+n)$$

$$S(m,n) = O(\log(m+n))$$

where $m = |\texttt{S1}|$ and $n = |\texttt{S2}|$.

Sorting using reduce

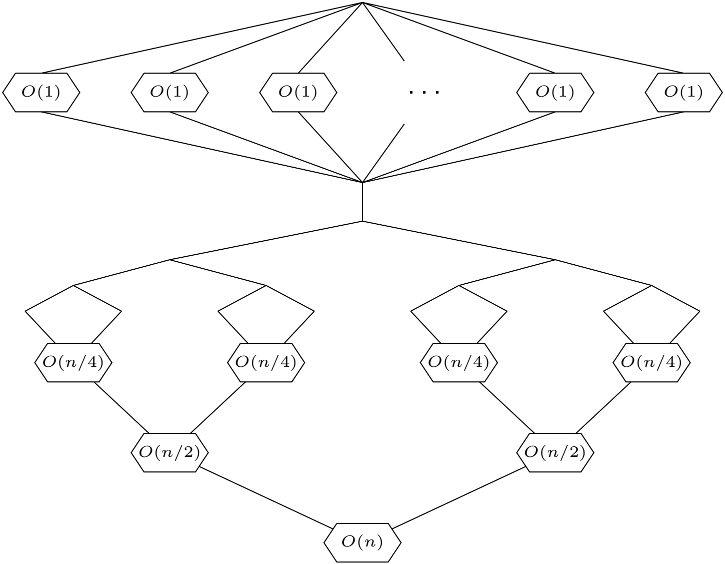
20.4

```
  val sort
    : ('a * 'a -> order) -> 'a seq -> 'a seq
  val merge
    : ('a * 'a -> order) -> 'a seq * 'a seq ->
    'a seq
```

20.8

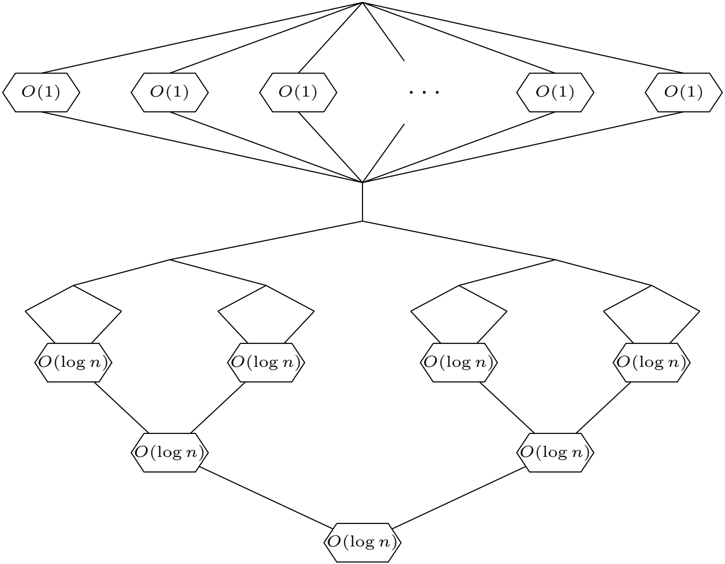
```
fun msort cmp S =
  let
    (* O(n) work, O(1) span *)
    val singletons =
      Seq.map Seq.singleton S
  in
    Seq.reduce (Seq.merge cmp)
      (Seq.empty ()) singletons
  end
```

Work



$$W = O(n \log n)$$

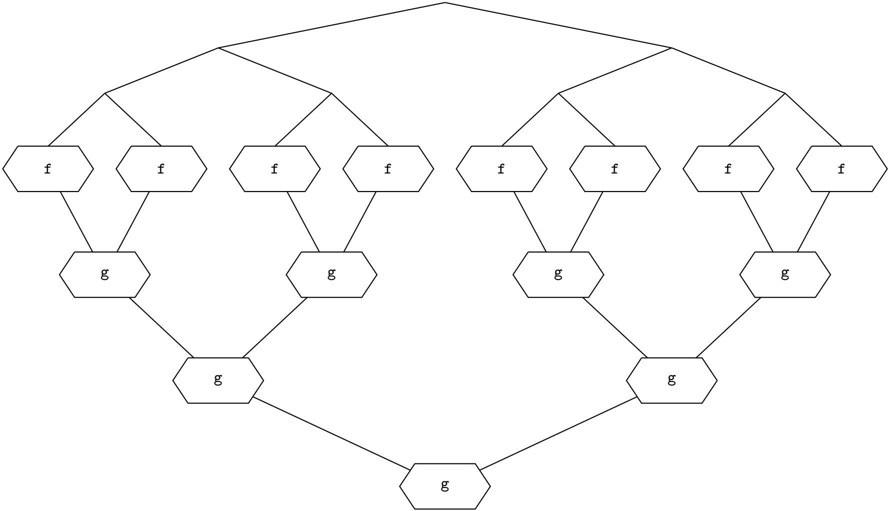
Span



$$S = O((\log n)^2)$$

mapreduce

`Seq.mapreduce f z g S`



Sorting using mapreduce

20.4

```
  val sort
    : ('a * 'a -> order) -> 'a seq -> 'a seq
  val merge
    : ('a * 'a -> order) -> 'a seq * 'a seq ->
    'a seq
```

20.9

```
fun msort cmp =
  Seq.mapreduce
    Seq.singleton
    (Seq.empty ())
    (Seq.merge cmp)
```

Section 2

Views

In the `SEQUENCE` signature, the following type is declared:

20.10

```
datatype 'a lview = Nil
                  | Cons of 'a * 'a seq
```

with the following values

20.11

```
val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq
```

Slow filter

20.12

```
fun filt1 p S =
    case (Seq.showl S) of
      Seq.Nil => Seq.empty ()
    | (Seq.Cons(x,xs)) =>
        if p x
        then Seq.hidel(
               Seq.Cons(x,filt1 p xs))
        else filt1 p xs
```

This has $O(n^2)$ work, and $O(n)$ span, assuming p is constant-time.

Sequential `filter`

```
fun filt1' p =
    Seq.fromList
    o (List.filter p)
    o (Seq.toList)
```

This has $O(n)$ work, and $O(n)$ span, assuming `p` is constant-time.

Sequences are like trees

In the `SEQUENCE` signature, the following type is declared:

20.13

```
datatype 'a tview = Empty
                  | Leaf of 'a
                  | Node of 'a seq * 'a seq
```

with the following values

20.14

```
val showt : 'a seq -> 'a tview
val hidet : 'a tview -> 'a seq
```

Parallel `filter`

20.15

```
fun filt2 p S =
   case (Seq.showt S) of
     Seq.Empty => Seq.empty ()
   | (Seq.Leaf x) => if p x
                     then Seq.singleton x
                     else Seq.empty ()
   | (Seq.Node(L,R)) =>
       Seq.hidet(
         Seq.Node(filt2 p L,filt2 p R)
     )
```

This has $O(n \log n)$ work, and $O(\log n)$ span, assuming `p` is constant-time.

Parallel `filter`

20.16

```
fun filt3 p =
  Seq.mapreduce Seq.singleton
    (Seq.empty()) Seq.append
end
```

This has $O(n \log n)$ work, and $O(\log n)$ span, assuming `p` is constant-time.

Can we have both?

Is there a way to implement `filter` with the best of all these implementations (i.e. $O(n)$ work and $O(\log n)$ span, assuming `p` is constant-time)?

Yes, you'll learn about it in 15-210. You may assume that this is how ours is implemented (the sequence reference gives these time bounds).

Thank you!